# SANDIA REPORT

SAND2004-5462
Unlimited Release
Printed October 2004

# Scalable Fault Tolerant Algorithms for Linear-Scaling Coupled-Cluster Electronic Structure Methods

Ida Nielsen
Curtis Janssen
Matt Leininger

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

SAND2004-5462
Unlimited Release
Printed October 2004

# Scalable Fault Tolerant Algorithms for Linear-Scaling Coupled-Cluster Electronic Structure Methods

Ida Nielsen
Scalable Computing Department
Sandia National Laboratories
P.O. Box 7011
Livermore, CA 94550
ibniels@ca.sandia.gov

Curtis Janssen
Scalable Computing Department
Sandia National Laboratories
P.O. Box 7011
Livermore, CA 94550
cljanss@ca.sandia.gov

Matt Leininger
Scalable Computing Department
Sandia National Laboratories
P.O. Box 7011
Livermore, CA 94550
mlleini@ca.sandia.gov

**Abstract**

By means of coupled-cluster theory, molecular properties can be computed with an accuracy often exceeding that of experiment. The high-degree polynomial scaling of the coupled-cluster method, however, remains a major obstacle

in the accurate theoretical treatment of mainstream chemical problems, despite tremendous progress in computer architectures. Although it has long been recognized that this super-linear scaling is non-physical, the development of efficient reduced-scaling algorithms for massively parallel computers has not been realized. We here present a locally correlated, reduced-scaling, massively parallel coupled-cluster algorithm. A sparse data representation for handling distributed, sparse multidimensional arrays has been implemented along with a set of generalized contraction routines capable of handling such arrays. The parallel implementation entails a coarse-grained parallelization, reducing interprocessor communication and distributing the largest data arrays but replicating as many arrays as possible without introducing memory bottlenecks. The performance of the algorithm is illustrated by several series of runs for glycine chains using a Linux cluster with an InfiniBand interconnect.

# Acknowledgment

We thank Prof. Daniel Crawford at Virginia Tech for valuable discussions. His pilot coupled-cluster program was very helpful for the development of our code. We also wish to thank Prof. Crawford's student Nick Russ, who implemented the single processor local (T) correction in our code.

# Contents



# Figures

# Scalable Fault Tolerant Algorithms for Linear-Scaling Coupled-Cluster Electronic Structure Methods

## 1 Introduction

Quantum chemistry, the application of quantum mechanics to molecular problems, has evolved into an indispensable tool for understanding chemical physics. With the rapid development of advanced computational hardware and algorithms, the field has blossomed into a major component of molecular science, and it is often possible to compute thermodynamic or spectroscopic properties of molecules with an accuracy exceeding that of even the best available experiments. However, a serious limitation of quantum chemistry is its poor extensibility: The most rigorous and reliable methods – those that carefully correlate the motion of the electrons – can be applied only to small molecules containing at most a few atoms. Larger molecules remain out of reach due to the high-degree polynomial computational scaling of these methods with molecular size. For example, an accurate computation of the minimum-energy structure for the amino acid valine (containing only eight non-hydrogen atoms) would require approximately one week on modern high-performance computational hardware with a state-of-the-art quantum chemical program. For the valine-valine dimer, however, a comparable calculation would require nearly *two years* to complete. This high-degree polynomial scaling behavior represents perhaps the greatest obstacle to the application of high-level quantum mechanics to mainstream chemical problems.

Great improvements have been made in the scaling of a number of simpler quantum chemical techniques (e.g., Hartree-Fock and density functional theory), but the inherent complexity of the higher-accuracy methods has slowed the development of reduced- or linear-scaling implementations. Furthermore, because high-accuracy approaches involve such extreme cost, they are currently applied only to relatively small molecules, where there can be little advantage from linear scaling formalisms; that is, the prefactor of the linear steps would be large enough to actually make the linear scaling approach more time consuming than a conventional calculation. However, tremendous progress in computer architectures now makes it possible to treat systems large enough to be beyond the crossover point where the linear scaling methods become faster than their conventional counterparts. This now makes it reasonable and necessary to address the nonphysical scaling of the correlated methods.

One method in particular has emerged as the best for high accuracy calculations:

coupled-cluster (CC) theory[2]. In the singles and doubles excitation version of this theory (CCSD), the amplitudes of the singles and doubles excitations from a reference wavefunction are the unknowns. Products of these amplitudes are used to approximate higher order excitations. This gives rise to a coupled system of nonlinear equations that must be solved iteratively. Other variants of the coupled cluster method exist, some including higher order excitations and others eliminating the singles through reference orbital rotations. The version that includes triples corrections perturbatively, CCSD(T), has proved to be a particularly effective predictor of molecular properties, often rivaling the accuracy of experiment.

Coupled-cluster theory, in common with all other quantum chemical models, uses molecular orbitals (MO's), simple functions that describe the motion of individual electrons. The most convenient choice of such functions are the so-called canonical MO's, for which the resulting mathematical equations assume a particularly simple form. Unfortunately, canonical MO's are also the source of the computational scaling problem: Due to their delocalized nature, they result in interactions between electrons on spatially distant parts of a large molecule. One route to overcoming this serious hindrance is the concept of "local correlation", originally explored by Pulay and Saebø[7]. They demonstrated that if one abandons canonical MO's and instead chooses a more localized orbital form, vast numbers of electronic wavefunction parameters become negligible and may thus be ignored. This approach is tantamount to correlating only the motions of the electrons on parts of the molecule that are in close spatial proximity. Thus far, the local correlation idea has been applied to many-body perturbation theory and to coupled-cluster theory with some success[3, 8, 9, 10].

In this work we have employed the local correlation idea of Pulay and Saebø in the context of coupled-cluster theory to develop a reduced scaling local CCSD(T) algorithm. A parallelization scheme for this algorithm has been worked out, and a massively parallel implementation realized for the major computational steps.

# 2 Local Coupled-Cluster Theory

We will employ the local formalism presented by Pulay and Saebø[5, 6]. In this formalism, the occupied orbitals are localized molecular orbitals, and the virtual orbitals are projected atomic orbitals. The occupied orbitals are localized by means of the Pipek-Mezey localization procedure[4], and the projection of the atomic orbitals entails projecting the atomic orbital space against the occupied orbital space to ensure orthogonality of the occupied and virtual spaces.

A key idea in the local correlation method is the use of orbital domains. A domain consists of a set of atomic orbitals, and an orbital domain $[i]$ is assigned to each occupied orbital $i$. The domain $[i]$ includes all atomic orbitals that are spatially close to the localized orbital $i$. To construct an orbital domain $[i]$, the Mulliken charges $q_A^i$

are first computed as follows

$$q_A^i = \sum_{\mu \in A} \sum_{\nu} C_{\mu i} C_{\nu i} S_{\mu \nu}, \tag{1}$$

where $A$ designates an atom, $\mu$, $\nu$ are atomic orbitals, $C_{\mu i}$ is a molecular orbital coefficient, and $S_{\mu\nu}$ is an element of the overlap matrix. The charges are then ordered in decreasing order, and atoms $A$ are included until the sum of the included charges $q_A^i$ exceedes some threshold, typically 1.8. The domain $[i]$ is then constructed as the union of all the atomic orbitals on the included atoms. The procedure is described in detail in Refs. [1] and [3]. When constructing the coupled-cluster wavefunction, excitations out of the occupied orbitals are only allowed into the virtual orbitals in the corresponding domains. Thus, only single excitations out of an orbital $i$ into the domain $[i]$ are included. For the double excitations, pair domains $[ij]$ are constructed for each occupied $ij$ pair by combining the domains $[i]$ and $[j]$, and excitations are allowed out of $ij$ into the $[ij]$ domain only. Analogously, triple excitations out of a triplet $ijk$ are allowed into only the corresponding domain $[ijk]$, obtained by combining the $[i]$, $[j]$, and $[k]$ domains. By including only these excitations and by restricting the number of $ij$ pairs and $ijk$ triplets that are included as well as by employing various screening protocols for the integrals, a linear scaling algorithm can be obtained. To limit the number of included $ij$ pairs, pairs may be classified as strong or weak, depending on the minimum distance $R_{ij}$ between any two atoms in the domains $[i]$ and $[j]$. For example, strong pairs may be defined as pairs for which $R_{ij}$ equals zero (i.e., $[i]$ and $[j]$ share an atom). A screening protocol leading to linear scaling has been described in detail in Ref. [9].

In the following, the indices $i, j, k, l$ denote occupied, localized molecular orbitals, and the indices $r, s, t, u$ represent projected atomic orbitals. For the two electron integrals, the Mulliken notation is employed. Elements of the Fock matrix are denoted as $f_{xy}$, where $x, y$ are indices that belong to either the localized occupied molecular orbital or projected atomic orbital space. Elements of the overlap matrix in the projected atomic orbital basis are designated $S_{us}$, and $\delta_{ij}$ represents the Kronecker delta. $t_r^i$ and $T_{rs}^{ij}$ represent elements of the single and double substitution arrays, respectively.

The local closed-shell coupled-cluster singles residual and doubles residual equations take the following form

$$v_r^i = s_r^i + S_{rs} \sum_k [(2T_{st}^{ik} - T_{st}^{ki}) r_t^k - \beta_{ki} t_s^k], \tag{2}$$

$$\begin{aligned} V_{rs}^{ij} &= K_{rs}^{ij} + \sum_{tu} C_{tu}^{ij} (rt|us) + \sum_t (ri|ts) t_t^j + \sum_t (sj|tr) t_t^i + G_{rs}^{ij} + G_{sr}^{ji} \\ &\quad + S_{rt} [\sum_{kl} (\alpha_{ij,kl} - \delta_{jl}\beta_{ki} - \delta_{ki}\beta_{lj}) C_{tu}^{kl}] S_{us}. \end{aligned} \tag{3}$$

The intermediates used in Eqs. 2 and 3 are defined as follows:

$$
\begin{aligned}
C_{tu}^{ij} &= T_{tu}^{ij} + t_t^i t_u^j, && (4)\\
\alpha_{ij,kl} &= (ik|jl) + \sum_{rs} C_{rs}^{ij}(sk|rl) + \sum_r [t_r^i(rk|lj) + t_r^j(rl|ki)], && (5)\\
\beta_{ki} &= f_{ki} + \sum_r f_{rk} t_r^i + \sum_l [\sum_{rs} L_{rs}^{kl} C_{sr}^{li} + t_r^l l_r^{lki}], && (6)\\
G_{rs}^{ij} &= S_{rt}\{t_t^i s_s^j - \sum_k t_t^k [\sum_{uv} C_{uv}^{ij}(su|vk) + (sj|ik) + \sum_u (sj|ku) t_u^i + \\
&\quad \sum_u (ik|su) t_u^j]^T + T_{tu}^{ij} X_{us} \\
&\quad + \sum_k [(2T_{tu}^{ik} - T_{tu}^{ki}) Y_{us}^{kj} - \frac{1}{2} T_{tu}^{ki} Z_{us}^{kj} - T_{tu}^{kj} Z_{us}^{ki}]\}, && (7)\\
r_r^k &= f_{rk} + \sum_l L_{rs}^{kl} t_s^l, && (8)\\
s_r^i &= f_{ri} + (f_{rs} - A_{sr}) t_s^i - S_{rs} \sum_{kl} T_{st}^{lk} l_t^{kli} \\
&\quad + \sum_k [2 \sum_{st} C_{st}^{ik}(rs|tk) - \sum_{st} C_{st}^{ki}(rs|tk) + \sum_s (2(ri|sk) - (rs|ik)) t_s^k], && (9)\\
A_{rs} &= S_{us} \sum_{kl} L_{rt}^{kl} T_{tu}^{lk}, && (10)\\
X_{rs} &= f_{rs} - A_{rs} + \sum_{kt} [2(rs|kt) t_t^k - (rk|ts) t_t^k] - S_{ts} \sum_k r_r^k t_t^k, && (11)\\
Y_{rs}^{kj} &= (rk|sj) - \frac{1}{2}(rs|kj) + \sum_t (rk|ts) t_t^j - \frac{1}{2} \sum_t (rs|kt) t_t^j \\
&\quad + \frac{1}{2} \sum_l [L_{rt}^{kl}(T_{tu}^{lj} + 2 t_t^l t_u^j - \frac{1}{2} T_{tu}^{jl} - t_t^j t_u^l) - l_r^{klj} t_u^l] S_{us} \\
&\quad + f_{rk} t_u^j S_{us}, && (12)\\
Z_{rs}^{kj} &= (rs|kj) + \sum_t (rs|kt|t_t^j - S_{us} \sum_l [(rl|tk)(\frac{1}{2} T_{tu}^{jl} + t_t^j t_u^l) + (rl|kj) t_u^l], && (13)\\
L_{rs}^{kl} &= 2(rk|sl) - (rl|sk), && (14)\\
l_{rl}^{ki} &= 2(rl|ki) - (rk|li). && (15)
\end{aligned}
$$

The CCSD correlation energy is computed from the two-electron integrals and the single and double substitution amplitudes

$$E_{\mathrm{CCSD}}^{\mathrm{corr}} = \sum_{ijrs} [2(ri|sj) - (rj|si)] C_{rs}^{ij} + 2 \sum_{ir} f_{ir} t_r^i. \quad (16)$$

The residual equations are a set of nonlinear equations, and they are solved iteratively. The equations are converged when the root mean square value of the single and double residuals arrays as well as the change in the energy are below selected thresholds.

The expression for the (T) correction, $E^{\rm corr}_{\rm (T)}$, to the LCCSD energy may be written as

$$E^{\rm corr}_{\rm (T)} = E^{\rm S}_{\rm (T)} + E^{\rm D}_{\rm (T)}, \tag{17}$$

where $E^{\rm S}_{\rm (T)}$ and $E^{\rm D}_{\rm (T)}$ are the contributions from the single and double substitution amplitudes, respectively, and may be expressed as

$$\begin{aligned} E^{\rm S}_{\rm (T)} &= \sum_{i\geq j\geq k} (2-\delta_{ij}-\delta_{jk})[\sum_{rstr'} t^i_{r'} S_{rr'}(js|kt) X^{ijk}_{rst} + \sum_{rsts'} t^j_{s'} S_{ss'}(ir|kt) X^{ijk}_{rst} \\ &\qquad + \sum_{rstt'} t^k_{t'} S_{tt'}(ir|js) X^{ijk}_{rst}] \end{aligned} \tag{18}$$

$$E^{\rm D}_{\rm (T)} = \sum_{i\geq j\geq k} (2-\delta_{ij}-\delta_{jk}) \sum_{rst} W^{ijk}_{rst} X^{ijk}_{rst}, \tag{19}$$

where $X^{ijk}_{rst}$ is defined in terms of the triples amplitudes

$$X^{ijk}_{rst} = 4T^{ijk}_{rst} - 2T^{ijk}_{rts} - 2T^{ijk}_{tsr} - 2T^{ijk}_{srt} + T^{ijk}_{trs} + T^{ijk}_{str}. \tag{20}$$

The triples amplitudes $T^{ijk}_{rst}$ are obtained by solving the equation

$$Q^{ijk}_{rst} + W^{ijk}_{rst} = R^{ijk}_{rst} = 0, \tag{21}$$

where $R^{ijk}_{rst}$ is the triples residual, and $Q^{ijk}_{rst}$ and $W^{ijk}_{rst}$ are defined as

$$\begin{aligned} Q^{ijk}_{rst} &= \sum_u \{\sum_{r's'} f_{tu} T^{ijk}_{r's'u'} S_{rr'} S_{ss'} + \sum_{r't'} f_{su} T^{ijk}_{r'ut'} S_{rr'} S_{tt'} + \sum_{s't'} f_{ru} T^{ijk}_{us't'} S_{ss'} S_{tt'}\} \\ &\quad - \sum_l \{\sum_{r's't'} [f_{kl} T^{ijl}_{r's't'} + f_{jl} T^{ilk}_{r's't'} + f_{il} T^{ljk}_{r's't'}] S_{rr'} S_{ss'} S_{tt'}\}, \end{aligned} \tag{22}$$

$$\begin{aligned} W^{ijk}_{rst} &= \sum_u \{\sum_{r'} (us|tk) T^{ij}_{r'u} S_{rr'} + \sum_{s'} (ur|tk) T^{ij}_{us'} S_{ss'} \\ &\quad + \sum_{r'} (ut|sj) T^{ik}_{r'u} S_{rr'} + \sum_{t'} (ur|sj) T^{ik}_{ut'} S_{tt'} + \sum_{s'} (ut|ri) T^{jk}_{s'u} S_{ss'} \\ &\quad + \sum_{t'} (us|ri) T^{jk}_{ut'} S_{tt'}\} \\ &\quad - \sum_l \{\sum_{r's'} [(lj|kt) T^{il}_{r's'} + (li|kt) T^{lj}_{r's'}] S_{rr'} S_{ss'} \\ &\quad + \sum_{r't'} [(lk|js) T^{il}_{r't'} + (li|js) T^{lk}_{r't'}] S_{rr'} S_{tt'} \\ &\quad + \sum_{s't'} [(lk|ir) T^{jl}_{s't'} + (lj|ir) T^{lk}_{s't'}] S_{ss'} S_{tt'}\}. \end{aligned} \tag{23}$$

Eq. 21 must be solved iteratively to get the full local (T) correction, but an approximation to the full (T) contribution, (T0) can be computed by a non-iterative scheme. In the non-iterative scheme, the off-diagonal elements of the occupied-occupied block of the Fock matrix are ignored in the last three terms of Eq. 22. Consequently, only amplitudes of the type $T^{ijk}_{xyz}$, where $x, y, z$ represent projected atomic orbitals, contribute to $Q^{ijk}_{rst}$. For each $ijk$ block, it is then possible to diagonalize the fock matrix in the corresponding triples domain, and Eq. 21 then reduces to

$$\begin{aligned} Q^{ijk,(T0)}_{abc} &= [f_{cc} + f_{bb} + f_{aa} - f_{kk} - f_{jj} - f_{ii}]T^{ijk}_{abc} \\ &= D^{ijk}_{abc}T^{ijk}_{abc}. \end{aligned} \tag{24}$$

where the indices $a, b, c$ represent pseudocanonical virtual orbitals in the space obtained by diagonalizing the fock matrix in the $[ijk]$ domain. The (T0) triples may then be computed, as in the canonical case, by the following expression

$$T^{ijk}_{abc(0)} = -W^{ijk}_{abc}/D^{ijk}_{abc}. \tag{25}$$

The amplitudes computed in this way are labeled with (0) to distinguish them from the triple amplitudes computed from the iterative solution of Eq. 21 without making approximations.

We then get two expressions for the local coupled-cluster energy including the triples correction

$$E^{\text{corr}}_{\text{LCCSD(T)}} = E^{\text{corr}}_{\text{LCCSD}} + E^{\text{corr}}_{(T)}, \tag{26}$$

$$E^{\text{corr}}_{\text{LCCSD(T0)}} = E^{\text{corr}}_{\text{LCCSD}} + E^{\text{corr}}_{(T0)}. \tag{27}$$

$E^{\text{corr}}_{(T)}$ and $E^{\text{corr}}_{(T0)}$ are the local triples corrections computed from the (T) and (T0) amplitudes, respectively, as described above.

# 3 Parallel Sparse Data Representation

The parallel sparse data representation was chosen to permit rapid prototyping, ease of use, and good performance. The primary type is the C++ generic class `Array`. `Array`'s primary template parameter is an integer, which gives the number of indices in the array. `Array` relies on a variety of auxiliary classes that describe blocks, distribution schemes, index ranges, etc. `Array`'s store the block descriptors for contributing blocks, and the data for each block. `Array`'s are capable of manipulating this data by performance copying of `Array`'s, summation of `Array`'s, binary products of `Array`'s (a contraction) to produce a third `Array`, and contractions that result in a

```
class Range {
  public:
    // Used to select blocking method.
    enum BlockingMethod {AtomBlocking, ShellBlocking,
                         FunctionBlocking, ExtentBlocking };
    // Distribute basis functions using the given BlockingMethod.
    Range(const sc::Ref<sc::GaussianBasisSet> &,
          BlockingMethod b = ShellBlocking, int blocksize = 1);
    // Constructor using a fixed block size.
    Range(int nindex, int block_size);
    // Constructor that uses the block sizes given in the argument.
    Range(const std::vector<int> & block_size);
    // Returns the dimension of this range.
    int nindex() const;
    // Returns the number of blocks.
    int nblock() const;
    // Returns the size for the given block.
    int block_size(int i) const;
};
```

**Figure 1.** The `Range` interface.

scalar. Parallelization is supported by members that convert replicated to distributed arrays, distributed to replicated arrays, and distributed arrays to a different distribution scheme. Helper classes are also employed to permit the use of the implicit summation convention in the C++ source code, greatly increasing the readability of the program. This section will discuss these arrays and their auxiliary classes in more detail.

## 3.1 Auxiliary Classes

Several auxiliary classes are needed to support the operations of the `Array` class. These classes are implemented in C++. The most important of these are outlined in this section. Simplified interfaces are presented, showing only the most important features.

A `Range` object represents a set of integers, [0, N). For example, one `Range` could represent the atomic orbitals and another could represent the molecular orbitals. These ranges are subdivided into user defined blocks. It is these blocks that determine the granularity of `Array`'s data storage and distribution. A simplified interface for `Range` is given in Figure 1.

`BlockInfo` stores information about a block of data, but not the data itself. This information includes the block numbers for each index. The interface for `BlockInfo` is given in Figure 2. The `BlockInfo` class has a single template parameter, `N`. This parameter gives the number of indices.

The map structures that maintain `Array`'s internal blocks require a comparison functor to establish an ordering relationship between `BlockInfo` objects. The internal

```
template <int N>
class BlockInfo {
  public:
    // Initialize the blocks to the values in the argument, v.  The
    // number of blocks assigned is the smaller of N and v.size().
    BlockInfo(const std::vector<bi_t> &v);
    // Initialize the blocks to the values in the argument, b. The IndexList,
    // l, specifies the order used to extract the indices from b.
    BlockInfo(const BlockInfo<N> &b, const IndexList &l);
    // Set/retrieve block numbers.
    bi_t &block(int i);
    // Compute the size of this block.
    unsigned int size(const Range *indices);
};
```

**Figure 2.** The `BlockInfo` interface.

```
template <int N>
class IndicesLess {
  public:
    bool operator() (const BlockInfo<N>&b1, const BlockInfo<N>&b2) const;
    int compare(const BlockInfo<N>&b1, const BlockInfo<N>&b2) const;
};
```

**Figure 3.** The `IndicesLess` interface.

`Array` manipulations require a particular ordering relationship. The first index is most significant using a simple comparison between the integer values. If the first indices are equal then the second indices are compared, and so on. This is done with the `IndicesLess` object that is outlined in Figure 3. Template specializations exist for `IndicesLess` and `BlockInfo` for `N` = 2, `N` = 3, and `N` = 4, to speed up comparison operations.

An `Index` is used in the symbolic notation for contractions. Its interface is shown in Figure 4.

```
    class Index {
      public:
        Index(const std::string &name);
        Index(int value);
        const std::string name() const;
        int value() const;
        bool has_value() const;
        bool has_name() const;
        bool operator == (const Index &i) const;
        void set_name(const std::string &name);
        void set_value(int value);
    };
```

**Figure 4.** The `Index` interface.

```
template <int N, class Compare = IndicesLess<N> >
class Array {
  public:
    // The underlying date structure containing the blocks.
    typedef std::multimap<BlockInfo<N>, double*, Compare > blockmap_t;
    // Array's with varying numbers of indices can be initialized.
    Array(const Range &i0, const Range &i1);
    // Adds block b.  The data is not allocated.
    blockmap_t::value_type &add_unallocated_block(const BlockInfo<N>&b);
    // Allocate storage for all blocks.
    void allocate_blocks();
    // These and related operators enable the use of shorthand
    // for Array manipulations.
    ContractPart<N> operator()(const Index &i1);
    ContractPart<N> operator()(const Index &i1, const Index &i2);
    // Accumulates partial contributions from all nodes and places the
    // results on all nodes.  The block map must be the same on all nodes.
    void parallel_accumulate(const sc::Ref<sc::MessageGrp> &grp);
    // Convert a distributed array to a replicated array.
    void replicated_from_distributed(const sc::Ref<sc::MessageGrp>&,
                                     const Array<N,Compare> &);
    // Change the block distribution pattern.
    void distributed_from_distributed(const sc::Ref<sc::MessageGrp>&,
                                      const BlockDistrib<N> &,
                                      Array<N,Compare> &,
                                      bool clear_source_array = false);
};
```

**Figure 5.** The `Array` interface.

## 3.2 The `Array` Class

An `Array` maps `BlockInfo` objects to each block's data. Two template parameters can be given: `N` and `Compare`. `N` is the number of indices in the array and `Compare` is comparison functor used to sort the blocks. The `Array` interface is shown in Figure 5. The `ContractPart` type that appears as a return type of `operator()` is used to implement the symbolic notation for contractions. It is not directly visible to users and will not be discussed further.

## 3.3 `Array` Contraction

The most important operation using an `Array` is a contraction. An example of such an operation is

$$R^{ij}_{\mu q} = \sum_p S_{p\mu} T^{ij}_{pq} \tag{28}$$

The `Array` class organizes $R$, $S$, and $T$ into blocks. In terms of blocks, Equation 28 can be written

$$\left[\bar{R}^{ij}_{\mu q}\right]^{i'j'}_{\mu' q'} = \sum_{p,p'} \left[\bar{S}_{p\mu}\right]_{p'\mu'} \left[\bar{T}^{ij}_{pq}\right]^{i'j'}_{p'q'} \tag{29}$$

The primed (element) indices are treated as if the arrays are ordinary, dense multi-dimensional arrays. The unprimed (block) indices only run over a subset of the possible values, according to the screening protocol employed on the `Array`. The contract in terms of these sparse `Array`'s is implemented as follows:

$$\begin{array}{l} \forall i,j,\mu,q \mid \bar{R}^{ij}_{\mu q} \neq 0 \\ \quad \forall p \mid \bar{S}_{p\mu} \neq 0, \bar{T}^{ij}_{pq} \neq 0 \\ \qquad \left[\bar{R}^{ij}_{\mu q}\right]^{i'j'}_{\mu' q'} \leftarrow \left[\bar{R}^{ij}_{\mu q}\right]^{i'j'}_{\mu' q'} + \sum_{p'} \left[\bar{S}_{p\mu}\right]_{p'\mu'} \left[\bar{T}^{ij}_{pq}\right]^{i'j'}_{p'q'} \end{array}$$

The code is implemented in such a way that it is not necessary to explicitly loop though all indices. The map structures drive the computation and are used to obtain lists of indices corresponding to nonzero blocks.

# 4 A Scalar Local CCSD(T) Algorithm

Before implementing the local coupled-cluster method, we have implemented a program to perform local second-order Møller-Plesset perturbation theory (MP2) computations so that the scaling behavior of the model can be studied and potential bottlenecks identified. Figure 6 illustrates the polynomial scaling of various steps in the program. The localization procedure is inherently $\mathrm{O}(n^3)$, but the procedure is fast and is not expected to become dominant even for larger computations. Reducing the scaling of this step is desirable but is beyond the scope of the present work. For the integral transformation, the scaling is nearly quadratic, whereas sub-quadratic scaling ($\mathrm{O}(n^{1.7})$) is obtained in the iterative procedure. It is possible to make each of these steps truly linear scaling by employing more advanced screening protocols, but these protocols involve a computational overhead that may not be offset by the improved scaling, and we have therefore chosen not to implement them, at least for the present. It is also expected that scaling will continue to improve as the system size increases. We emphasize that the precise scaling of the algorithm is irrelevant as long as the scaling is reduced sufficiently to enable computations for the system sizes of interest.

The essential merit of reduced scaling correlation methods is the possibility of doing computations for larger systems than those that can be treated by conventional correlation methods. The touchstone for evaluating the merits of reduced scaling

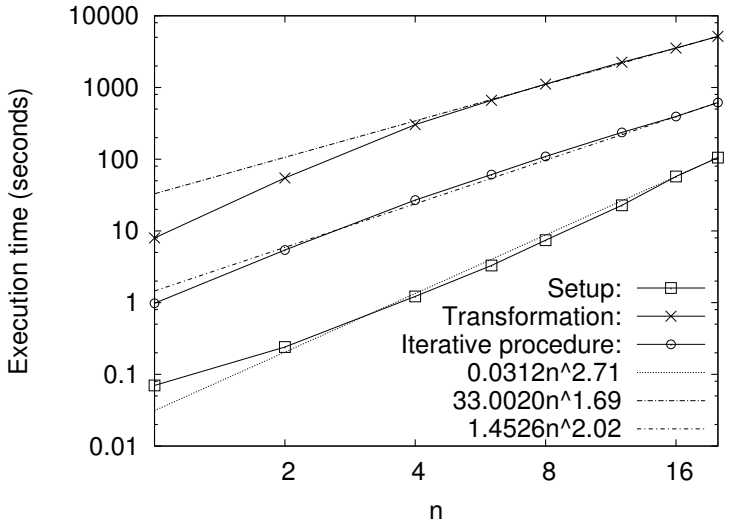


**Figure 6.** Execution time for $(\text{glycine})_n$ chains using a STO-3G basis set. For the iterative procedure, timings are given for one iteration. For each solid curve, a line representing the asymptote, computed from the last two points, is given.

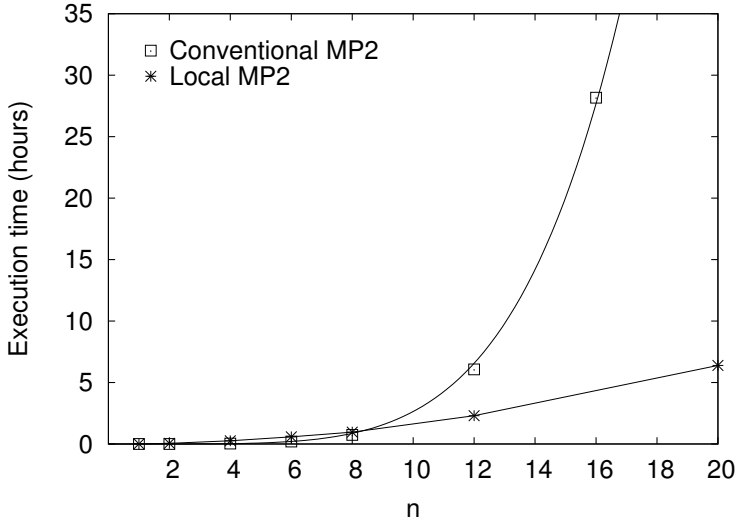


**Figure 7.** Comparison of timings for conventional and local MP2 computations for $(\text{glycine})_n$ chains using a 3-21G basis.

correlation procedures, therefore, is their performance relative to the corresponding conventional correlation procedures. As documented in Figure 7, we have performed

crossover studies for glycine chains comparing our reduced scaling local MP2 program and a conventional MP2 program. As expected, the conventional algorithm is faster for small systems, but the relative performance of the local MP2 algorithm improves rapidly with the size of the system, and the local MP2 method becomes faster than the conventional model around $(\text{glycine})_8$. For the larger glycine chains, the conventional method, due to its high computational scaling ($\text{O}(n^5)$) soon becomes impractical, whereas computations for the large glycine chains, e.g., $(\text{glycine})_{20}$ can easily be performed with the local MP2 method even on a single processor. The storage requirements for the local MP2 model are similarly reduced so that computations that become prohibitively expensive in terms of memory requirements with the conventional method still can be performed with the local MP2 model.

After initial studies of the local correlation model using the local MP2 program, a CCSD(T) algorithm was developed and implemented. An outline of the local CCSD(T) algorithm is shown in Fig. 8. The major computational steps in the algorithm are the transformation of the two-electron integrals and the computation of the doubles and triples residuals in the CCSD and (T) iterative procedures.

The transformation of the two-electron integrals is carried out as two separate transformations. The first transformation generates the integrals $(rs|ti)$ with one occupied index and the all-virtual integrals $(rs|tu)$, and in the second transformation, the integrals $(rs|ij)$, $(ri|sj)$, $(ri|jk)$, and $(ij|kl)$ with two, three, or four occupied indices are computed.

The local CCSD energy is computed in an iterative procedure as described in Chapter 2. After completion of the CCSD computation, the (T0) energy can be computed from the local CCSD single and double substitution amplitudes. The (T0) amplitudes are subsequently used as the initial guess for the triple substitution amplitudes in the iterative procedure computing the full (T) correction.

# 5 Parallel Implementation and Performance

All major computational steps in the algorithm have been parallelized. Where possible, parallelization has been implemented by distributing pairs $(ij)$ or triplets $(ijk)$ of localized occupied molecular orbitals and letting each node process only the subset of the $ij$ pairs or $ijk$ triplets held locally. For cases where parallelization over $ij$ or $ijk$ is not possible, work has been distributed over a single index representing either an occupied orbital or an atomic orbital. The employed data distribution scheme distributes all the largest data arrays but replicates as many data arrays as possible without introducing memory bottlenecks. The replication of most data arrays greatly simplifies the parallelization of the various contraction steps that constitute the major computational steps in the computation of the coupled-cluster energy and also reduces the amount of communication required. The largest data arrays, representing the atomic

Initialize all arrays
Compute screening quantities required for integral transformations

Integral transformation: generate $(rs|tu)$, $(rs|ti)$
Integral transformation: generate $(rs|ij)$, $(ri|sj)$, $(ri|jk)$, $(ij|kl)$

Begin CCSD iterations
    Compute singles residual
    Compute doubles residual
    Compute $\Delta\mathbf{t}$, $\Delta\mathbf{T}$
    DIIS extrapolation of $\mathbf{t}$, $\mathbf{T}$
    Compute $E^{\rm corr}_{\rm CCSD}$, $\Delta E^{\rm corr}_{\rm CCSD}$
    Check for convergence on $\Delta E^{\rm corr}_{\rm CCSD}$, $\Delta\mathbf{t}$, $\Delta\mathbf{T}$
End CCSD iterations

Compute (T0) correction

Begin (T) iterations
    Compute triples residual
    Compute $\Delta\mathbf{T3}$
    Compute $E^{\rm corr}_{(T)}$
    Check for convergence on $\Delta E^{\rm corr}_{\rm (T)}$, $\Delta\mathbf{T3}$
End (T) iterations

**Figure 8.** Outline of the scalar local CCSD(T) algorithm. The arrays $\mathbf{t}$, $\mathbf{T}$, and $\mathbf{T3}$ represent the single, double, and triple substitution amplitudes, respectively.

orbital basis and partially transformed two-electron integrals, are distributed across nodes, but all other arrays, including the fully transformed two-electron integrals, the singles and doubles residuals and the single and double substitution amplitudes, are replicated. Let us briefly describe how the major computational steps, including the two-electron integral transformation and the computation of the doubles and triples residuals, are parallelized.

The two-electron integral transformations (there are two transformations, *vide supra*) are implemented using a coarse-grained parallelization strategy to reduce communication, and the same parallelization scheme is employed for both transformations. An outline of the part of the transformation generating the $(rs|tu)$ integrals is shown in Fig. 9. All partially transformed integral arrays are distributed, whereas the fully transformed integrals are replicated across nodes. The partially transformed integral arrays are distributed by distributing an atomic orbital index ($\mu$ in Fig. 9). With this distribtion, the first three quarter transformations can be performed independently on each node without any interprocessor communication. When the third

$(\bar{\mu}\rho|\nu u) \leftarrow (\bar{\mu}\rho|\nu\sigma) \times P(\sigma, u)$ First quarter transformation
$(\bar{\mu}\rho|tu) \leftarrow (\bar{\mu}\rho|\nu u) \times P(\nu, t)$ Second quarter transformation
$(\bar{\mu}s|tu) \leftarrow (\bar{\mu}\rho|tu) \times P(\rho, s)$ Third quarter transformation
$(\mu s|t\bar{u}) \leftarrow (\bar{\mu}s|tu)$ **Redistribution of integrals**
$(rs|t\bar{u}) \leftarrow (\mu st|\bar{u}) \times P(\mu, r)$ Fourth quarter transformation
$(rs|tu) \leftarrow (rs|t\bar{u})$ **Global accumulation of transformed integrals**

**Figure 9.** The integral transformation generating the $(rs|tu)$ integrals. A horizontal bar above an index indicates that the index is distributed. The steps requiring communication are printed in bold face.

quarter transformation is complete, a redistribution of the three-quarter transformed integrals is required, generating the three-quarter transformed integrals distributed over the index $u$ representing a projected atomic orbital. The fourth quarter transformation, subsequently, is performed on each node without any further communication. After completion of the fourth quarter transformation, the tranformed integrals held locally on each node are sent to all other nodes in a global accumulation step.

Computation of the doubles residual (Eq. 3) is parallelized by distributing work over $ij$ pairs. Each node computes the elements $V_{rs}^{ij}$ of the residual for a subset of the $ij$ pairs, and no communication is required until the computation of the residual is complete on each node. At this point, a global accumulation is carried out to send the computed residuals from each node to all other nodes.

The computation of the triples residual (Eq. 21) is parallelized in a similar way, except that triplets $ijk$, not $ij$ pairs, are distributed. Each node independently computes a subset of the triple residual elements $R_{rst}^{ijk}$, and, afterwords, the computed residuals are accumulated on all nodes.

The parallel performance of the algorithm is illustrated in Figs. 10 and 11. These curves were obtained on a Linux cluster with dual 3.06 GHz XEON nodes and a 4X InfiniBand interconnect. Only one processor per node was used for these runs. Performing only a local MP2 computation (which is a subset of the coupled-cluster computation), speedups were obtained for $(\text{glycine})_4$ using the 6-31G and 6-31G* basis sets. Due to the distributed memory requirements for these algorithms, the 6-31G basis set computation would run on 4 or more nodes and the the 6-31G* basis set computation required at least 16 nodes. The dominant computational step is the 4-index integral transformation, and the speedups for the total time closely track the integral transformation speedup. Steady improvement is seen as the number of nodes increases, with a drop-off from ideal behaviour seen around 8-16 processors. Larger problem sizes are expected to increase the number of processors where a significant

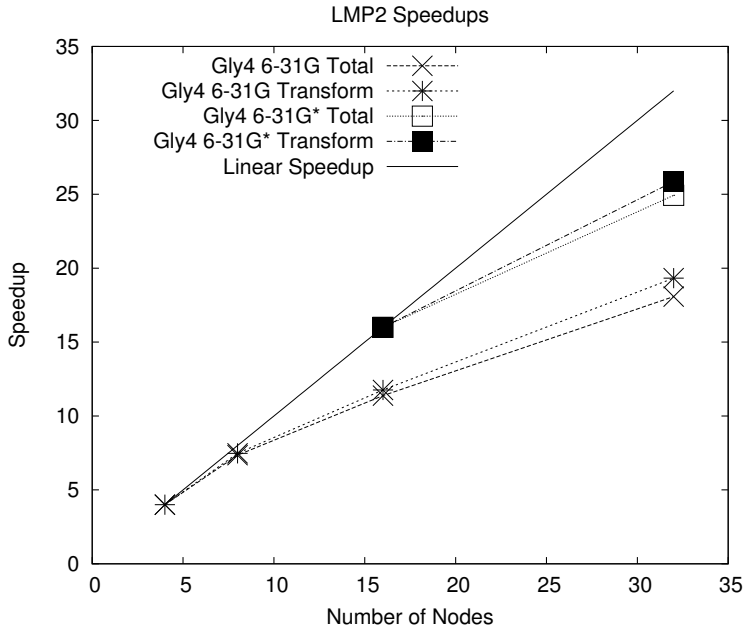


**Figure 10.** Parallel speedups for the LMP2 total time and transformation step for $(\text{glycine})_4$ using 6-31G and 6-31G* basis sets.

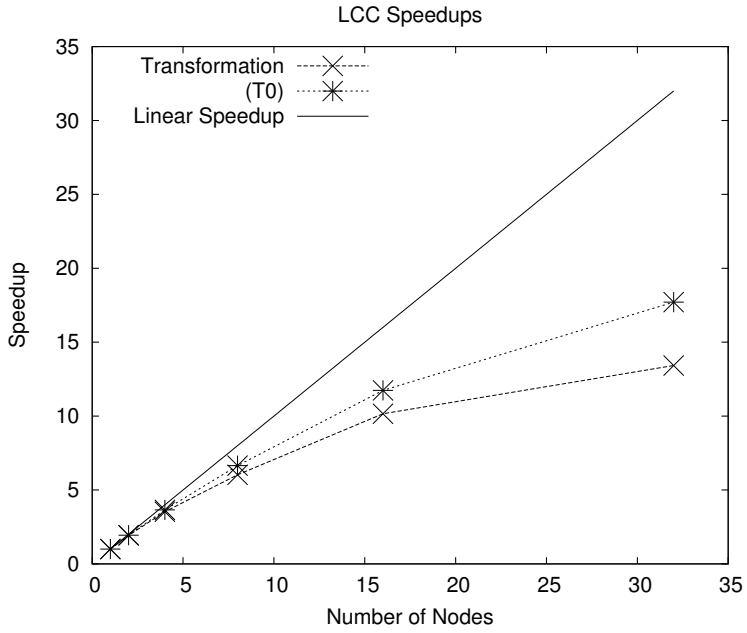


**Figure 11.** Parallel speedups for the LCCSD transformation and (T) steps for $(\text{glycine})_2$ using a 6-31G basis set.

deviation from ideal before is first seen.

The LCCSD(T0) speedups were obtained for $(\text{glycine})_2$ using the 6-31G basis set. The steps shown are the 4-index transformation (which for LCCSD produces a larger set of integrals than the LMP2 algorithm needs to compute) and the (T0) correction. The performance trends are very similar to those for the LMP2 speedups. The LCCSD code is currently memory bound, which limited the size of the problems that could be studied. In this case, the LCCSD iterative procedure obtains a low parallel efficiency ($\sim$6% for 32 nodes).

The replicated all-virtual integrals $(rs|tu)$ and integrals with one occupied index $(rs|ti)$ currently constitute memory bottlenecks in the program. The $(rs|tu)$ integrals only appear in one term in the computation of the doubles residual, and this term can be computed in an integral-direct manner, making it possible to avoid storage of the $(sr|tu)$ integrals. Regarding the $(rs|ti)$ integrals, the memory bottleneck may be avoided by distributing the integrals across nodes.

# 6 Fault Tolerance

A major goal of this work has been to understand how the new parallel algorithms described herein can be implemented in a way that will permit application handled scalable fault tolerance. We define scalable fault tolerance as having the following two properties: The time-to-solution in the absence of faults is not significantly different than the time-to-solution of an algorithm that does not support fault tolerance and the time-to-solution in the presence of faults is not significantly different than the time-to-solution when running a algorithm that is not fault-tolerant on the number of processors remaining after the fault. Handling faults directly in the application has several advantages in that only the application programmer knows at any given time what data is hard to recompute and what data is easy to recompute in case of a failure. That knowledge can for many applications dramatically reduce the expense of providing fault tolerance compared to automatic checkpointing. The disadvantages of handling faults within the application is that this requires correct processing during uncommon circumstances, which creates additional burdens in developing the algorithm and in providing regression tests that guarantee that correct results are obtained in the presence of faults.

The local electronic structure theories that we are investigating provide a good testbed for fault tolerant algorithms in that they exhibit a range of properties and thus would capture the behavior of a wide variety of other applications. It has turned out that the first phase of the calculation, the four index transformation of the two electron integrals, is very data intensive and requires that most of the data be distributed among the nodes. The second phase, the iterative solution of the LMP2 equations in the localized basis, is compute intensive and requires relatively small amounts of

data. In this phase it is best to replicate the data on all nodes.

Fault handling in the second phase would be the simplest. At the beginning of the iterative solution a record can be made of which processor is responsible for each occupied orbital pair. These records can be replicated on each task. At the end of each iteration, when the updated solution is redistributed, the new iterate can be accumulated into a temporary buffer. The application will then check for a fault. If no fault occurs, then the temporary iterate is copied into the current iterate and the calculation proceeds. If a fault has occurred, then a new communicator can be built from the remaining processors and processing for that iteration can resume by redistributing the work from the faulty tasks onto the remaining nodes to complete the iteration. At the application level, the processing time will be exactly the same obtained by starting the iteration with the number of nodes remaining after the fault. The only performance penalty will be a result of the need for the MPI layer to quiesce all message passing and purge messages to and from the faulty nodes.

Fault handling in the first phase is more interesting as it requires a way to handle the loss of distributed data. The large amount of data involved makes checkpointing to a filesystem undesirable. Total replication of the data is impossible because more data is involved than can be held on any node. Recomputation of the lost data is complicated because of the unavoidable redistribution of third quarter transformed integrals. After this step, each node will contain contributions from all other nodes. If any node's data is lost, then the entire set of second transformed integrals must be recomputed to recompute the lost third quarter transformed integrals. This leaves two alternatives: Partial replication or storing the second quarter transformed integrals until the fourth quarter transformed integrals are completed. Our current thinking is that partial replication is the best approach as this would reduce the the application code complexity somewhat.

The actual implementation of fault tolerance has been hampered by the slow development of MPI extensions that report failures to the application and give the application an opportunity to continue running. Based on this and the feedback from the second year review of this project suggesting that we focus on the electronic structure methods, we have not gone beyond the above analysis of fault tolerance in our application. We continue to work with investigators at Sandia and other labs to provide the MPI features required for application level fault tolerance.

# 7 Conclusions

We have developed a local, reduced-scaling coupled-cluster algorithm for computing CCSD(T) energies. The algorithm also computes local MP2 energies as well as the CCSD(T0) energy, which is an approximation to the full local CCSD(T) energy. A preliminary implementation of the MP2 part of the local coupled-cluster algorithm

method exhibited low-order polynomial scaling and demonstrated very significant computational savings relative to a conventional MP2 computation. For the local coupled-cluster algorithm, a parallelization scheme has been developed and implemented for massively parallel computation of local CCSD(T) energies. The parallel performance of both the MP2 part and the coupled-cluster part of the algorithm has been investigated.

Parallel performance testing has yet to be done for larger cases where the benefits of both the local correlation scheme and of the parallel implementation will be more pronounced. Currently, the replicated storage of two-electron integrals of the type $(rs|tu)$ and $(rs|ti)$ constitute a memory bottleneck, but this bottleneck may be removed relatively easily by modifying the algorithm to no longer require storage of the $(rs|tu)$ integrals and to distribute the $(rs|ti)$ integrals.

Finally, strategies for implementing fault tolerance have been investigated. Whereas fault handling is relatively straightforward in the part of the code that employs replicated data arrays, a more sophisticated fault handling scheme involving partial replication of data is required for the part of the code that employs distributed data.

# References


[1] J. W. Boughton and P. Pulay. *J. Comp. Chem.*, 14:736, 1993.

[2] T. D. Crawford and H. F. Schaefer. *Rev. Comp. Chem.*, 14:33, 2000.

[3] C. Hampel and H.-J. Werner. *J. Chem. Phys.*, 104:6286, 1996.

[4] J. Pipek and P. G. Mezey. *J. Chem. Phys.*, 90:4916, 1989.

[5] S. Saebø and P. Pulay. *J. Chem. Phys.*, 86:914, 1987.

[6] S. Saebo and P. Pulay. *J. Chem. Phys.*, 88:1884, 1988.

[7] S. Saebø and P. Pulay. *Ann. Rev. Phys. Chem.*, 44:213, 1993.

[8] M. Schütz, G. Hetzer, and H.-J. Werner. *J. Chem. Phys.*, 111:5691, 1999.

[9] M. Schütz and H.-J. Werner. *J. Chem. Phys.*, 114:661, 2001.

[10] G. E. Scuseria and P. Y. Ayala. *J. Chem. Phys.*, 111:8330, 1999.

DISTRIBUTION:

1 MS 9158
Ida Nielsen, 8961

1 MS 9158
Curtis Janssen, 8961

1 MS 9158
Matt Leininger, 8961

3 MS 9018
Central Technical Files, 8940-1

1 MS 0899
Technical Library, 9616

1 MS 9021
Classification Office, 8511, for
Technical Library, MS 0899,
9616 DOE/OSTI via URL

1 MS 0188
D. Chavez, LDRD Office, 1030